(19)日本国特許庁（ＪＰ）　(12) **公開特許公報（Ａ）**　(11)特許出願公開番号

特開平10－116302

(43)公開日　平成10年(1998)５月６日

(51)Int.Cl.[6]	識別記号	ＦＩ	
Ｇ０６Ｆ　17/50		Ｇ０６Ｆ　15/60	６５４Ａ
			６５４Ｄ
			６５４Ｇ

審査請求　未請求　請求項の数15　ＯＬ　（全 19 頁）

(21)出願番号　特願平9－249152

(22)出願日　平成９年(1997)９月12日

(31)優先権主張番号　９６１９０９６．２

(32)優先日　1996年９月12日

(33)優先権主張国　イギリス（ＧＢ）

(71)出願人　000005049
シャープ株式会社
大阪府大阪市阿倍野区長池町22番22号

(72)発明者　アンドリュー　ケイ
イギリス国　オーエックス４　１エイチエイ　オックスフォード，　ハースト　ストリート　99

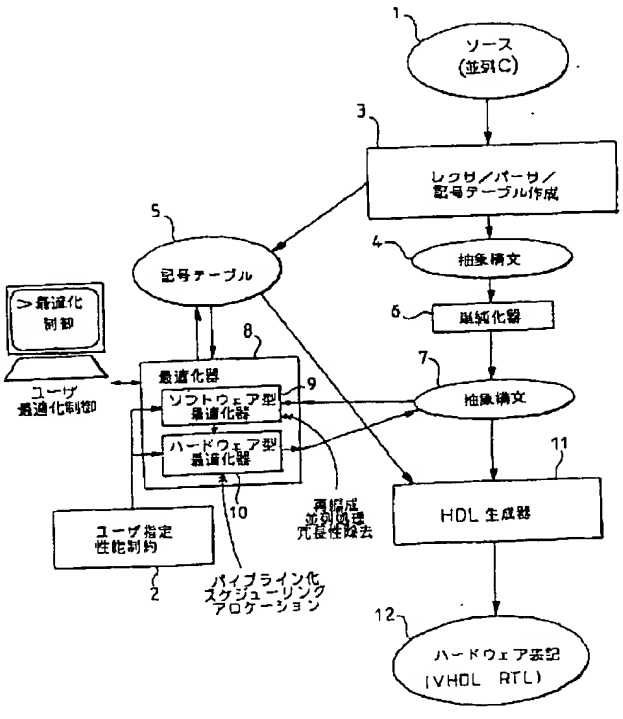
(74)代理人　弁理士　山本　秀策

(54)【発明の名称】　集積回路の設計方法及びそれによって設計された集積回路

(57)【要約】

【課題】　高レベル言語で記述されたソフトウェアレベルからハードウェアレベルへの変換時間を短縮し、ハードウェア開発の効率を向上させる集積回路の設計方法を提供する。

【解決手段】　並列処理及び同期通信をサポートするプログラミング言語でその機能を定義することによって、集積回路が設計される。得られたソースコード（１）が、コンパイラに与えられる。コンパイラは、集積回路の外部通信の順番を変えずに同期通信のタイミングを変える最適化器モジュール（８）を含む。コンパイラは、集積回路の回路構成を表す出力コード（１２）を生成する。出力コード（１２）は、統合ツール、及び集積回路の製造における後工程にも提供され得る。



1

【特許請求の範囲】

【請求項１】　集積回路の機能を並列処理及び同期通信をサポートするプログラミング言語で定義するステップと、

該集積回路の外部通信の順番を変えずに同期通信のタイミングを変えるように構成されたコンパイラを適用して、該集積回路の回路構成を表す出力コードを生成するステップと、を包含する、集積回路の設計方法。

【請求項２】　前記同期通信がハンドシェークを含む、請求項１に記載の集積回路の設計方法。

【請求項３】　前記コンパイラが、抽象構文木及び記号テーブルを形成するように構成されている、請求項１或いは２に記載の集積回路の設計方法。

【請求項４】　前記コンパイラが、前記抽象構文木を単純化するソフトウェア最適化器を含む、請求項３に記載の集積回路の設計方法。

【請求項５】　前記ソフトウェア最適化器が、各コンポーネント部分に変数が１つ存在するように複合データ構造をコンポーネント部分に変換するように構成されている、請求項４に記載の集積回路の設計方法。

【請求項６】　前記ソフトウェア最適化器が、未使用変数を除去するように構成されている、請求項４或いは５に記載の集積回路の設計方法。

【請求項７】　前記ソフトウェア最適化器が、ループ外の共通演算子を移動させるように構成されている、請求項４から６のいずれか一つに記載の集積回路の設計方法。

【請求項８】　前記コンパイラが、前記出力コードによって表記されるハードウェアインプリメンテーションを最適化するハードウェア最適化器を含む、請求項１から７のいずれか一つに記載の集積回路の設計方法。

【請求項９】　前記ハードウェア最適化器が、スケジューリング及びアロケーションを行うように構成されている、請求項８に記載の集積回路の設計方法。

【請求項１０】　前記コンパイラが、少なくとも１つの所定の性能パラメータが達成されたときに最適化を終了するように構成されている、請求項４から９のいずれか一つに記載の集積回路の設計方法。

【請求項１１】　前記少なくとも１つの所定の性能パラメータが、集積回路の最大面積を含む、請求項１０に記載の集積回路の設計方法。

【請求項１２】　前記少なくとも１つの所定の性能パラメータが、集積回路の最小処理速度を含む、請求項１０或いは１１に記載の集積回路の設計方法。

【請求項１３】　前記少なくとも１つの所定の性能パラメータが、最大消費パワーを含む、請求項１０から１２のいずれか一つに記載の集積回路の設計方法。

【請求項１４】　前記出力コードで定義される構成を実行する回路構成を表すレジスタ転送レベルコードを生成するステップをさらに含む、請求項１から１３のいずれか一つに記載の集積回路の設計方法。

【請求項１５】　請求項１から１４のいずれか一つに記載の方法によって設計された集積回路。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】本発明は、集積回路の設計方法に関する。本発明はまた、そのような方法によって設計された集積回路に関する。

【０００２】

【従来の技術】大規模集積（ＬＳＩ）回路のデザインは、例えばＡＮＤ、ＯＲ、ＮＯＴ、ＦＬＩＰ－ＦＬＯＰ等の２進機能を実行するゲートの集合体と、それらのゲートの相互接続に関する仕様と、を含む。その後に、デザインを適切な技術での製造に適した形式に変換するために、レイアウトツールが使用され得る。

【０００３】このようなデザインを作成する公知の技術では、「概略的な獲得（schematiccapture)」の名で知られる方法が用いられる。この技術によれば、ユーザは、グラフィックソフトウェアツールを用いて、ライブラリから得た各論理ゲート或いはゲートの集合体を配置し、コンピュータのマウスを用いて配線を「描く」ことによってこれらのゲートを相互接続することができる。その後に、例えばゲートを除去或いは単純化することによって、回路の全体機能を変えずに得られた回路を最適化し、これをレイアウト及び製造工程に出すことができる。しかし、設計者は、全て或いは殆ど全てのゲート或いはゲートの集合体についてのタイミング及び論理を考慮しなければならない。従って、この技術は、大規模な設計に使用することが難しく、またエラーを生じ易い。

【０００４】別の公知の技術では、設計者が、ＬＳＩ回路の記述をハードウェア記述言語（ＨＤＬ）で書く。ＨＤＬの各ステートメントは最終デザインにおける数個のゲートに対応するので、最終デザインにおける論理の複雑さに比べれば入力ソースコードは比較的に短い。従って、設計者の生産性が向上する。公知のＨＤＬには、IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993, IEEE, New York, 1993に開示されるＶＨＤＬ、及びD.E. Thomas and P.R. MoorbyによりThe Verilog Hardware Description Language, Kluwer Academic 1995に開示されるＶｅｒｉｌｏｇがある。このような言語をS. CarlsonによりIntroductionto HDL-Based Design Using VHDL, Synopsys Inc., CA, 1991（文献１）に開示されるような適切な統合ツールとともに用いることにより、デザインを回路構成に変換する。

【０００５】このようなＨＤＬを用いた統合技術を利用して新たなＬＳＩ回路を設計する際には、その回路の挙動についてのアルゴリズムが、ソフトウェア技術者によって、Ｃ言語として知られるような適切な高レベル言語でキャプチャされる。その後に、例えばＣ言語で書かれた「テストハーネス」を用いて、アルゴリズムの挙動が

正しいかどうかを調べるテストを行う。テストハーネスは、回路シミュレータ或いはエミュレータを用いて、その回路設計がテストされ得る環境を記述する。ディスク或いはランダムアクセスメモリ（ＲＡＭ）に格納された回路に対しては、ベクトルとして知られている入力セットを利用したテストをコンパイルして且つ実行するために、標準的なコンパイラを有するワークステーションが用いられる。

【０００６】次のステップでは、文献１に開示されるＶＨＤＬレジスタ転送レベル（ＲＴＬ）のような、ハードウェアの統合及びシミュレーションにより適した言語に、ハードウェア技術者がＣコードを書き換える。この時点では、どの種のアーキテクチャを用いるか、データをパイプライン化するかどうか、外部に対する回路インターフェースをどのようにするか、及び各ストラクチャに何ビット分の格納場所をアロケートするのか等、多数の設計上の選択肢が存在する。典型的には、ＶＨＤＬバージョンは、オリジナルのＣバージョンに比べて、大きさが１桁大きくなる。

【０００７】ＣバージョンとＨＤＬバージョンとの間に直接的な結び付きがないので、ＨＤＬの記述にエラーが生じることがあり、そのために、この段階でテストを行うことが必要不可欠である場合が多い。デザインがテストされ得る前には、例えばＶＨＤＬなどで新しいテストハーネスを書かなければならない。このハーネスもまた、Ｃ言語で書かれたハーネスに比べて１桁大きくなる場合が多い。ＶＨＤＬバージョンを綿密にテストした後には、上記のような適切な統合ツールを用いて、回路に変換することができる。しかし、回路に統合され得るＶＨＤＬ構成のセットは、ＶＨＤＬ言語全体のサイズと比べて比較的に小さい。また、タイミング及びアーキテクチャに関する決定の殆どは、ユーザによって明示的に注釈されなければならず、従って、ユーザは個々の言語構成がどのように統合されるかに関してかなり詳細な知識を持っていなければならない。この知識は、異なる統合ツール間では異なる。

【０００８】この時点で、統合された回路が、所望のデザインに対して遅過ぎたり大き過ぎたりすることを発見することができる。そのような場合、ＨＤＬを調節してデザインを指定の範囲内に戻すことができる場合もあるが、それができない場合には、Ｃ言語で書かれた新たなアルゴリズムを試作する必要があり、設計時間が長くかかってしまう。

【０００９】例えばD. Gajski, N. Dutt, A. Wu and S. LinによってHigh-Level Synthesis, Introduction to Chip and System Design, Klewer, 1992（文献２）に開示されているような高レベルハードウェア設計言語を提供するために、ＨＤＬの抽象化レベルの向上が図られている。その一例に、Synopsys On-Line documentation 3.2b (CDROM format), Synopsys Inc., CA, 1995に開示されているシノプシス・ビヘイビア・コンパイラがある。このコンパイラは、「挙動性(behavoural)」ＶＨＤＬで書かれたソースコードを受け取り、比較的低レベルな統合可能ＶＨＤＬの出力を生成する。この入力言語は、標準的な統合可能サブセットよりも広いフルＶＨＤＬ言語のサブセットから派生する。コンパイラは、そのデザインのアーキテクチャを選択し、マイクロプロセッサコアとしてそれをモデリングし、回路全体の速度要件に見合う十分なハードウェアが利用可能であるようにする。このコンパイラは最適化を提供し、文献２に開示されるようなスケジューリング及びアロケーション方式アルゴリズムによって速度と面積とのトレードオフを行う。

【００１０】しかし、依然としてユーザは、クロック端がどこで生じるのかを注釈することによってタイミング情報を提供し、且つ、どのクロックサイクルで入力及び出力データが利用可能でなければならないのかを知っている必要がある。このような理由から、このシステムを使用しようとする設計者には、ハードウェアに関する相当な知識が要求される。また、得られるハードウェア記述は、オリジナルの挙動性ＶＨＤＬ記述とは挙動が異なるので、２つの異なるテストハーネスが必要となり得る。さらに、このシステムは、そのタイミング要件への依存性のために、アルゴリズムの試作には適さない。なぜなら、これらは、現在はクロックサイクルレベルであり、サブクロックレベルではないからである。

【００１１】他の公知のコンパイラとしては、I. Page and W. LuckによりCompiling Occaminto FPGAs, 271-283, Abingdon EE & CS books, 1991に開示されているようなＨａｎｄｅｌコンパイラ及びＨａｎｄｅｌ－Ｃコンパイラがある。Ｈａｎｄｅｌコンパイラは、例えばInmos, The Occam 2 Programming Manual, Prentice-HallInternational, 1988に開示されるような、Ｏｃｃａｍの名で知られる言語で書かれたソースコードを受け取る。ＯｃｃａｍはＣに似た言語であるが、並列処理及び名前付きチャネルを介した同期２点間通信を表現するための余分な構成を有する。Ｈａｎｄｅｌ－Ｃコンパイラも殆ど同じであるが、ソース言語が若干異なっており、Ｃ言語に慣れたプログラマにとってより馴染み易いものとなっている。

【００１２】コンパイラが並列構成を提供するので、プログラマは、設計上の問題に対する可能な解決策として、並列アルゴリズムを考えることができる。プログラマがどのサイクルで開始しようともメッセージが全く失われないことを確実にするために、周知のタイプの単純な「ハンドシェーク」技術によって同期通信が行われる。従って、送信側及び受信側の両者は、通信が完了するのを待ってからでなければ、続行することができない。言語によってこの制約が課せられるので、プログラマが通信イベントを再スケジューリングする際の自由度が増す。例えば、プログラマが、ｃ１及びｃ２と名付け

られたチャネルを介してそれぞれ１０及び２３という値の送信を要求する場合、受信プロセスが適切に書かれているとすれば、上記データは、どの順番でも、並列にも送信され得て、或いは、ｓｅｎｄコマンドの前とｓｅｎｄコマンドの間とに任意の遅延を伴っても送信され得る。このための疑似コードの一例は、

【００１３】

【数１】

```
seq[send(c1,10);send(c2,23);]
OR seq[send(c2,23);send(c1,10);]
OR par[send(c1,10);send(c2,23);]
OR seq[delay(x);send(c1,10);delay(y);send(c2,23);]
```

【００１４】のようになる。

【００１５】ハンドシェークプロトコルは、（どのような形態で実施されていても）受信側が準備状態にあるときにデータアイテムが必ず受信され且つ全く失われないことを確実にする。このようにして、コンパイルされた回路の２つの部分が正確にいつ相互作用するかに関して、幾分の自由度がある。

【００１６】しかし、Ｈａｎｄｅｌの場合は、（通信以外の）各構成のタイミングの総合的な制御は、プログラマが行う。各構成には、正確なサイクル数が割当てられている（これを、タイムドセマンティクスと呼ぶ）。従って、プログラマは、デザインにおける全ての低レベル並列処理を考慮しなければならず、且つ、コンパイラが各構成をどのようにクロックサイクルに割当てるのかを知っていなければならない。例えば、プログラマは、

【００１７】

【数２】

```
a:=b*c+d*e
```

【００１８】を指定することができる。しかし、全てのアサインメントに１サイクルしか要しないので、両方の乗算１サイクルで行われる必要がある。これは、２つの乗算器が形成されなければならないことを意味し、面積が大きくなる。また、これらの乗算器が単一のサイクルで動作しなければならないので、クロック速度が遅くなる。

【００１９】さらに、主としてタイムドセマンティクスのために、Ｈａｎｄｅｌが対応できない重要な構成がいくつかある。そのような構成には、あるアレイ（ＲＡＭ）を２回参照するアサインメント（なぜなら、これは高価なデュアルポートＲＡＭを示唆する）、ファンクションコールを含む表現、ならびにパラメータ付き関数が含まれる。

【００２０】

【発明が解決しようとする課題】タイムドソース言語セマンティクスを用いる公知の言語は、オリジナルのソースコードにおける暗示的なタイミングに従ってしまう。

従って、コンパイラがプログラムの実行に要するサイクル数を変えるような最適化を行うことができないことがあって、オリジナルソースコードにおけるタイミング挙動をエンコードするのがユーザの責任になる。従って、タイムドセマンティクスを利用する場合には、設計者自身が最適化を行わなければならず、コンパイラがそれを助けることはできないので、設計時間の点で問題がある。

【００２１】本発明は、上記課題を克服するためになされたものであって、その目的は、（１）高レベル言語で記述されたソフトウェアレベルからハードウェアレベルへの変換時間を短縮し、ハードウェア開発の効率を向上させる集積回路の設計方法を提供すること、及び（２）そのような設計方法に従って設計された集積回路を提供すること、である。

【００２２】

【課題を解決するための手段】本発明の第１の局面によれば、集積回路の機能を並列処理及び同期通信をサポートするプログラミング言語で定義するステップと、該集積回路の外部通信の順番を変えずに同期通信のタイミングを変えるように構成されたコンパイラを適用して、該集積回路の回路構成を表す出力コードを生成するステップと、を包含する集積回路の設計方法が提供され、そのことによって上記目的が達成される。

【００２３】前記同期通信は、ハンドシェークを含み得る。

【００２４】前記コンパイラは、抽象構文木及び記号テーブルを形成するように構成され得る。前記コンパイラは、前記抽象構文木を単純化するソフトウェア最適化器を含み得る。前記ソフトウェア最適化器は、各コンポーネント部分に変数が１つ存在するように複合データ構造をコンポーネント部分に変換するように構成され得る。前記ソフトウェア最適化器は、未使用変数を除去するように構成され得る。前記ソフトウェア最適化器は、ループ外の共通演算子を移動させるように構成され得る。

【００２５】前記コンパイラは、前記出力コードによって表記されるハードウェアインプリメンテーションを最適化するハードウェア最適化器を含み得る。前記ハードウェア最適化器は、スケジューリング及びアロケーションを行うように構成され得る。

【００２６】前記コンパイラは、少なくとも１つの所定の性能パラメータが達成されたときに最適化を終了するように構成され得る。前記少なくとも１つの所定の性能パラメータは、集積回路の最大面積を含み得る。前記少なくとも１つの所定の性能パラメータは、集積回路の最小処理速度を含み得る。前記少なくとも１つの所定の性能パラメータは、最大消費パワーを含み得る。

【００２７】該方法は、前記出力コードで定義される構成を実行する回路構成を表すレジスタ転送レベルコードを生成するステップをさらに含み得る。

【００２８】本発明の第２の局面によれば、本発明の第１の局面による方法によって設計された集積回路が提供され、そのことによって上記目的が達成される。

【００２９】以下に、本発明の作用を説明する。

【００３０】ハンドシェーク等の通信プロトコルを高レベルの最適化と共に使用すれば、コンパイラは効率的なインプリメンテーションをもたらし得るので、通信を抽象的に表現する際のより高い自由度が設計者にもたらされる。入力言語は、高レベルで且つプログラマに馴染み易いものであり得るとともに、ハードウェアにおいて認識可能な表記を有する重要な構成の殆どをサポートし得る。言語は、並列処理及び２点間通信を表現し得るが、タイムドセマンティクスを持たない。コンパイラは、例えばソースコードレベルに近い比較的高レベルで最適化を行うことができるとともに、ＨＤＬを出力することができる。従って、低レベルの統合、最適化及びハードウェアマッピングは、業界標準ツールを用いて行うことができる。並列構成及び２点間通信を伴うＣ言語に類似したソフトウェア言語を用いたデザインの機能は、効率的なＬＳＩ設計のために自動的に或いは半自動的にＨＤＬに変換される。アンタイムドソース言語セマンティクスを使用すれば、コンパイラによって、オリジナルソース言語の仕様に従いながらデザインのタイミングを向上させる最適化を行うことが可能になる。

【００３１】

【発明の実施の形態】以下に、添付の図面を参照しながら本発明の実施例を説明する。

【００３２】図１に示されるコンパイラは、「並列Ｃ言語」の名で知られる高レベル言語で書かれたソースコード１を受け取る。この言語は、構成とハンドシェーク２点間通信プリミティブとを含み、例えば最終デザインのコスト及び性能に関するユーザ指定性能制約（参照符号２で図示）を指定する。コンパイラは、モジュール３を有する。モジュール３は、入力ソースコードの構文解析及びチェックを行うことにより、中間抽象構文木（ＡＳＴ）表記４及び記号テーブル５を作成する。記号テーブル５は、ソースコードによって宣言された名前及びオブジェクトを記録するものである。A. V. Aho and J. D. UllmanによりPrinciples of Compiler Design, Addison-Wesley, 1977（文献３）第１９７～２４４頁に、これに適したレクサ及びパーサが開示されている。中間構文は、シーケンシャル、並列、及びループ構造のための表記を有し、また、演算のスケジューリング及びアロケーションを表すために用いられる特定の注釈を有する。

【００３３】ＡＳＴは、単純化器モジュール６に与えられる。単純化器モジュール６は、全ての不体裁な構成を、よりシンプルな構成に書き直す。特に、後のステージをより単純にコード化できるように、単純化器モジュール６からの出力はフル抽象構文のサブセットだけが使用されている。例えば、単純化器モジュール６は、複合データ構造を、各コンポーネントに変数が１つであるコンポーネント部分に分解する。

【００３４】単純化されたＡＳＴ７は、最適化器モジュール８に与えられる。最適化器モジュール８は、ソフトウェア最適化器モジュール９とその次にあるハードウェア最適化器モジュール１０とを含んでいる。ソフトウェア最適化器モジュール９は、例えば文献３の第４０６～５１７頁に開示されるような、不使用変数の除去及びループ外の共通処理の移動のようなソフトウェア最適化技術を用いて、単純化されたＡＳＴ７を最適化する。ハンドシェークのようなプロトコルを用いて通信を行うので、転送中にデータが失われることがない。従って、デザインにおける最終タイミングに影響を及ぼし得るが、そのような通信の時間的順序は変えないように、単純化されたＡＳＴ内で通信の移動を行うことができる。並列処理を許容する言語でソースコード１が書かれているため、ソフトウェア最適化器モジュール９は、性能制約２を満たすように計算をパイプライン化する等の手段を導入することができる。

【００３５】全ての最適化が設計の向上に貢献するように、推定関数が用いられる。推定関数を用いて、ゲート数、回路面積、回路速度、待ち時間、スループット、消費パワー、リソース要件などを推定することができる。ソフトウェア最適化が完了すると、ハードウェア最適化器モジュール１０は、ハードウェアターゲットに固有の最適化を行う。文献２の第１３７～２９６頁に、適切なハードウェア最適化器が開示されている。回路面積及びゲート数はハードウェアの作成における重要な考慮点であるので、上記の最適化は、可能な限りハードウェアが再利用され得るように、タイミングを考慮して設計される。このプロセスには、パイプライン化、スケジューリング、及びアロケーションの技術が含まれ、各最適化に対して再び推定関数を行うことにより、確実に改善がなされているようにする。モジュール１０による最適化の結果によって、各変数及び演算子の最適なハードウェア表記についての追加情報が、記号テーブル５に追加される。例えば、初期化された後に書込みが全く行われないアレイは、読出し／書込みアレイに必要なＲＡＭよりも安価なリードオンリーメモリ（ＲＯＭ）によって表すことができる。

【００３６】デザインが、ユーザによって指定された性能及びコスト制約２を満たしていることが推定関数によって示されたとき、モジュール９及び１０による最適化が終了し得る。これらの制約を満たすことが不可能な場合、ユーザにメッセージが与えられ得る。さらに、特定の最適化を選択する、或いは、最適化のパラメータを与えることによって、最適化器モジュール８に命令を与えるためのユーザインタラクションがさらに設けられても良い。この最適化は、適切な環境下におけるデザインの機能性には影響を与えないが、これによって、異なる性

能或いはコストを達成し得る。

【００３７】最適化されたＡＳＴ及び改変された記号テーブルは、ＨＤＬ生成器モジュール１１に与えられる。ＨＤＬ生成器モジュール１１は、ＡＳＴを詳細に検討し、そして、記号テーブル５に集められた情報を用いて各構成についてのハードウェア表記を作成する。このようにして、生成器モジュール１１は、ＶＨＤＬ ＲＴＬ等の適切な言語で書かれたハードウェア表記１２を提供する。その後、業界標準ツールを用いて、ＲＴＬをＬＳＩ回路に変換することができる。

【００３８】モジュール８が行い得る最適化の種類の３つの例を、以下に説明する。

【００３９】（例１）この例は、ソフトウェア式の最適化によってアルゴリズム性能を向上させる１つの方法を示す。

【００４０】

【数３】

```
a := 3
b := 1
while (true)
    b := b + (a * a)
    send (ch, b)
```

【００４１】というコードを考える。この例に適用可能な最適化がいくつかある。まず、ａ＊ａはループの反復の度に計算されるが、ａの値はループ内で変わらないので、答は常に同じである。ループ開始前に行う１回の計算と１回の一時的変数の代入とによって、これを置換し得る。これは、ハードウェア上では、ループの実行中に乗算器を開放して他の場所での乗算器の使用を可能にするとともに、乗算器が２サイクル以上のサイクルを要求する場合にはループ待ち時間を短縮する可能性もある。これが可能であるのは、ｓｅｎｄコマンドがタイミングに依存しないからであり、

【００４２】

【数４】

```
a := 3
b := 1
tmp := a*a
while (true)
    b := b+tmp
    send (ch, b)
```

【００４３】のようになる。

【００４４】２番目の最適化は、ａ＝３及びｔｍｐ＝９とすることである。従って、このプログラムは、

【００４５】

【数５】

```
a := 3
b := 1
while (true)
    b := b+9
    send (ch, b)
```

【００４６】のように書くことができる。

【００４７】次に、ａは一度も読み出されないので、

【００４８】

【数６】

```
b := 1
while (true)
    b := b+9
    send (ch, b)
```

【００４９】のように、ａを取ってしまうことが可能である。

【００５０】（例２）ハードウェア式の最適化の例として、

【００５１】

【数７】

```
a = b*c + d*e
```

【００５２】というアサインメントを考える。

【００５３】先に述べたように、公知のコンパイラにおけるタイムドセマンティクスでは、やはり、ソース言語で書かれたこのステートメントは、実行時には、１クロックサイクルで実行されなければならない。これは、２つのフラッシュ乗算器及び１つの加算器を設ける他には、あまり選択の余地がない。結果的に、２つの乗算器のために面積が大きくなり、サイクル時間が遅くなる（フラッシュ乗算器は、典型的に、入力の幅に依存して大きな組合せ遅延を有する）。

【００５４】しかし、図１のコンパイラには、上記のような制限が全くない。上記の乗算は、

【００５５】

【数８】

```
a := b*c
a := a + d*e
```

【００５６】と示すようにシーケンシャル化することができる。

【００５７】次に、

【００５８】

【数９】

```
a := sys_mult(b,c)
a := a + sys_mult(d,e)
```

【００５９】に示すように、上記乗算を共有乗算関数に代入することが可能である。

【００６０】乗算器は、その時の処理の種類に合わせることができる（高速ではあるが大型なパラレル乗算、または、数サイクルを必要とはするが非常に小型で且つクロック速度の速いシーケンシャル乗算）。いずれの場合も、乗算器は１つで十分である。実際の選択は、自動的になされても、或いは、コンピュータのガイダンスによってユーザが行ってもよい。

【００６１】（例３）この例では、最終的なデザインの効率を向上させるために、どのようにアサインメントの再スケジューリングを行うかを示す。

【００６２】

【数１０】

```
a = b*c;
c = x*y;
send (output, a);
b = a+d;
```

【００６３】というプログラム例を考える。

【００６４】乗算器と加算器とが１つずつある場合、受信側がａの値を受信する準備ができていれば、上記プログラム全体を、

【００６５】

【数１１】

```
a = b*c;
par
{
c = x*y;
send (output, a);
b = a+d;
}
```

【００６６】のように、２サイクルに圧縮できる。

【００６７】公知のコンパイラでは、通信をこのように再スケジューリングすることができないので、この最適化を行うことはできない。例えば、タイムドセマンティクスを有するコンパイラでは、ソース言語のタイミングを変えることはできず、一方、挙動性コンパイラ(Behavioral Compiler)のようなコンパイラでは、通信によって課せられる境界を越えて最適化を行うことはできない。

【００６８】あるデザインのソースコード１は、Ｃ言語のサブセットにいくつかの追加を含む並列Ｃ言語で書かれている。追加分は、以下の通りである。

【００６９】（ａ）並列処理のための構成：ｐａｒ

ｐａｒ構成は、プログラム内のどこにでも使用することができ、これにより、システムレベルから単一ステートメントレベルまでのあらゆる細分性の並列処理が導入され得る。この構文は、

【００７０】

【数１２】

```
statement = ...
          | par {[statement]*}
```

【００７１】に示すように、通常のＣステートメント構文を拡張する。

【００７２】例えば、

【００７３】

【数１３】

```
par {
    func1(x,y);
    func2(x+y,z);
    }
```

【００７４】のようにして、２つのファンクションコールを並列に実行する。

【００７５】（ｂ）所与のタイプの同期チャネル

これらのチャネルにより、１つのｐａｒにおけるブランチ間での通信、及び、（Ｃ言語の通常の外部キーワードとともに用いられる場合に）同期回路とその周辺との間での通信が可能になる。内部チャネルは、

【００７６】

【数１４】

```
declaration = ...
            | [storage_class] chan [type]identifier[,identifier]*;
```

【００７７】に示す宣言の構文の拡張を用いて宣言される。

【００７８】例えば、

【００７９】

【数１５】

```
chan struct comm a, b;
```

【００８０】のようにすれば、構造成タイプｃｏｍｍのデータを用いて通信する２つの内部チャネルａ及びｂが宣言される。

【００８１】チャネルは単方向性であるので、周辺との通信を行うには、そのプロセスが、任意の共有チャネルの送信端或いは受信端のいずれを有しているのかが分かっている必要がある。これは、キーワードｃｈａｎｉｎ及びｃｈａｎｏｕｔによって区別されるので、全ての外部チャネル宣言において、これらのキーワードを使用しなければならない。

【００８２】

【数１６】

```
declaration = ...
            | [storage_class] chanin [type] identifier [identifier]*;
            | [storage_class] chanout [type] identifier [,identifier]*;
```

【００８３】例えば、

【００８４】

【数１７】

```
extern chanin int from_env;
extern chanout int to_env;
```

【００８５】のようにして、１６ビットの整数で周辺と通信するチャネルｆｒｏｍ＿ｅｎｖ及びｔｏ＿ｅｎｖを宣言する。

【００８６】（ｃ）プリミティブｓｅｎｄ（ｃｈａｎ、ｖａｌ）及びｒｅｃｅｉｖｅ（ｃｈａｎ）

ｓｅｎｄ（ｃｈａｎ、ｖａｌ）は、チャネルｃｈａｎを介して値ｖａｌを送信する。ｒｅｃｅｉｖｅ（ｃｈａｎ）は、チャネルｃｈａｎを介して値を受信するものであり、表現を作成する際に使用できる。各チャネルは２点間方式でデータの通信を行い、通信を行っている２つの処理はそれぞれ、通信が完了するのを待ってからでなければ続行できない。さらに、チャネルｃｈａｎを介して送信されるのを待っているデータが存在するときに真となる関数ｒｅａｄｙ（ｃｈａｎ）がある。

【００８７】

【数１８】

```
statement = ...
          | send (identifier, expression);
expression = ...
          | receive (identifier)
          | ready (identifier)
```

【００８８】次の例は一対のプロセスを示し、一方のプロセスは、整数を生成してそれを他方のプロセスに（チャネルｃｈを用いて）送信し、この他方のプロセスは、受信した整数を加算する。

【００８９】

【数１９】

```
chan int ch;
par {
    {
        int i = 0;
        while (1)
            send (ch, i++);
    }
    {
        int tot = 0;
        while (1)
            tot += receive(ch);
    }
}
```

【００９０】（ｄ）所与のビット幅の整数型のセット

これは、どのような数値精度が要求された場合でも、効率的な回路が形成できるようにするためのものである。このために、＃ｅが含まれるように、型修飾子のセットが拡張される。ここで、ｅは、ｅの値に等しい幅を示す定数表現である。

【００９１】

【数２０】

```
type_modifier = ...
              | #constant_expression
```

【００９２】例えば、

【００９３】

【数２１】

```
chan unsigned#7 c;
```

【００９４】のようにして、「符号無し７ビット整数」型のｃと呼ばれるチャネルを宣言する。

【００９５】（ｅ）ビット操作を行う効率的な回路を構築するためのビット選択及びビット連結演算子

記号@は、連結を表す。「ｇｒａｂ」演算子（<－と書く）は、表現ｅ、及び定数ビット位置$b_1 \ldots b_n$のリストを要する。この演算子が評価されると、$e_{b1} \ldots e_{bn}$のｎビットの結果が返される。但し、e_iは、ｅのｉ番目のビットである。

【００９６】

【数２２】

15　　16

```
expression = ...
           | expression @ expression
           | expression<_{constant_expression[,constant_expression]*}
```

【００９７】例えば、３ビット２進数では5_{10}＝１０１$_2$であり、7_{10}＝１１１$_2$である。従って、６ビット２進数では、5_{10}@7_{10}＝１０１１１１$_2$＝47_{10}である。47_{10}から上位４ビットを選択すると、表現47_{10}<－{５，４，３，２}は、値１０１１$_2$＝11_{10}を生成する。

【００９８】入力言語の標準Ｃ部分は、ｉｆ、ｗｈｉｌｅ、ｓｗｉｔｃｈ、ｂｌｏｃｋｓ、ｆｕｎｃｔｉｏｎｓ等の全ての制御特徴、ならびに、ポインタを除く演算及びデータ操作の殆ど全てを有する。しかし、アレイのインデックスを用いてポインタを真似ることは可能である。回路の外部のＲＡＭ或いはＲＯＭコンポーネントであると仮定される「外部アレイ」を除いて、アレイは、統合回路内の専用ロジックとして実現される。

【００９９】上記言語におけるＣ言語部分のセマンティクスは、Ｃ言語のセマンティクス（つまり、ｅｘｐｒｅｓｓｉｏｎｓ、ａｓｓｉｇｎｍｅｎｔ、ｉｆ、ｗｈｉｌｅ、ｆｏｒ、ｂｒｅａｋ等）と類似している。ｐａｒ及びチャネル通信のセマンティクスは、上記ＩＮＭＯＳの文献に開示されるＯｃｃａｍのセマンティクス、及び、C.A.R. HoareによってCommunication Sequential Processes, InternationalSeries in Computer Science, Prentice-Hall, 1985に開示されるＣＳＰのセマンティクスと類似している。Ｏｃｃａｍの用法ルールに類似する用法ルールがある。２つの異なる並列コンポーネントから同一の変数がアクセスされる場合、そのアクセスが全てリードオンリーでない限り、挙動は不確定である。

【０１００】先に説明したように、ソースコードがファイルに入力された後、図１の３においてコンパイラはコンパイルを開始し、標準的な構文解析技術を用いてソースコードを構文解析して、デザインの構造やサブ構造などを記録する抽象構文木４とするとともに、使用される全ての識別子のタイプ及び名前を記録する記号テーブルを作成する。処理が進むと、記号テーブルは、各識別子についての情報を照合することによって様々な変換ステージを互いに関連付ける。

【０１０１】次の工程は、図１の６において抽象構文木を単純化することである。これを行うのは、生成器１１によって、プログラミング特徴の全てをハードウェアに変換することはできないからである。単純化器モジュール６は、それらのサポートされない構成を除去して、ＨＤＬ生成器モジュール１１によってサポートされる等価な構成に置き換える。例えば、生成器モジュールは、標準的なＣ言語におけるａ＝（ｂ＋＋）＋５のように、アサインメントが副次的効果を有することを許可しない。単純化された等価物は、ａ＝ｂ＋５；ｂ＝ｂ＋１のようになり得る。さらに、単純化器モジュール６は、全ての演算子及び定数の幅と型とを計算し、この情報を構文木に格納する。

【０１０２】ｓｅｎｄ（ｃｈ、Ｒ）は、ｃｈ：＝Ｒのようなアサインメントに単純化される。この表記は、記号テーブルにおけるｃｈの型によって、それが本当はチャネル送信であることを「知る」。しかし、この表記法の統一性（どのプロトコルが要求される場合でも、デスティネーションは常にアサインメントの左側に書かれる）のために、後のトランスフォーメーションが殆ど例外無く記述される。同様に、ｘ：＝ｒｅｃｅｉｖｅ（ｃｈ）は、ｘ：＝ｃｈというアサインメントに単純化される。

【０１０３】この時点で、デザインは、単純化された抽象構文７及び記号テーブル５の組合せで表現される。ＨＤＬ生成器が処理できない全ての構成を取り去り、最適化を行う必要がある。例えば、外部アレイ（ＲＡＭ）へのアクセスは、逆の情報がない限り、シングルポートＲＡＭであると想定される。従って、ｍｅｍ［ｉ］：＝ｍｅｍ［ｊ］のような表現は、ＨＤＬ生成器によって正しく処理されない。なぜなら、ＨＤＬ生成器は、そのメモリへの２つのアクセスを（ほぼ）同時に生成するからである。この表現は、ｌｏｃａｌ　ｔ；ｔ：＝ｍｅｍ［ｊ］；ｍｅｍ［ｉ］：＝ｔ；のように書き換えられる。

【０１０４】標準的な最適化の１つの可能な方法は、ループ内のある計算を反復する必要がない場合に、その計算をループから取り去ってしまうことである。例えば、

【０１０５】

【数２３】

```
While (x<10)
  x  := x + (y*y)
```

【０１０６】は、

【０１０７】

【数２４】

```
declare tmp
tmp: = y*y
While (x<10)
  x  := x + tmp
```

【０１０８】のように書き換えられる。

【０１０９】もう１つの可能な方法は、寿命時間が重ならない複数の変数の間でレジスタを共有することである。例えば、

【０１１０】

【数２５】

```
Sequence:
Declare tmp1
    tmp1 := f(x)
    send(ch,tmp1)
Declare tmp2
    tmp2 := g(y)
    send (ch, tmp2)
```

【０１１１】は、

【０１１２】

【数２６】

```
Sequence:
Declare tmp
tmp := f(x)
send(ch,tmp)
tmp := g(y)
send(ch,tmp)
```

【０１１３】のように書き換えられる。

【０１１４】最適化器モジュール１０が、ある特別な場合にしか用いられない構造を見つけた場合、生成器モジュール１１がその情報を利用してより簡潔なコードを生成できるように、最適化器モジュール１０は、構文木（或いは記号テーブル）にその情報を記録することができる。例えば、あるアレイが定数によって初期化され、その後に全く更新されない場合、より高価なＲＡＭよりも安価なＲＯＭとして、そのアレイが実現され得る。

【０１１５】抽象構文における１つの重要な構成は、アサインメント同期化の形態である。これにより、数個のアサインメントを同時に実行して、時間若しくは格納スペース或いはそれら両方を節減することができる。例えば、ｌｏｃａｌ　ｔｍｐ；ｔｍｐ：＝ａ；ａ：＝ｂ；ｂ：＝ｔｍｐ；とする代わりに、これを最適化して、ｓｙｎｃｈ｛ａ：＝ｂ　ＡＮＤ　ｂ：＝ａ｝とすることが可能である。アサインメントは、レジスタ、チャネル、及びアレイの値の全ての通信を処理する。この構造により、いくつかの有用なアクションを短縮し、短縮しなかった場合よりも時間を短くすることが可能になる。何が同期化され得るかについては制限があり、その制限は、インプリメンテーションに依存する。

【０１１６】最適化器モジュール１０は、さらに根本的なことをすることが可能であり、例えば、シーケンシャルコードをパラレルに動作させる、或いは、その逆を行う、チャネルを除去して、より弱い形態の同期化に置き換える、インラインに関数を拡張する、乗算を共有乗算関数へのコールにすることにより乗算器を共有する、コード或いは表現の重複部分を共有する、そして、得られる回路の外部挙動が変わらない場合に複雑な表現をパイプライン化することができる。無論、速度、面積及びサイクル数は変化し得るが、外部インターフェースは全てハンドシェークを有するので、通信の順序が守られる場合は、このような変化は影響しない。

【０１１７】最適化は、自動的に適用されてもよいし、ユーザ命令型もしくはユーザ選択型であってもよい。目標は、特定の用途によって決まる指定の面積、パワー、或いは時間の範囲を達成することである。これらの属性は、抽象表記に適用される単純なメトリクスによって推定され得る。

【０１１８】ＨＤＬ生成器モジュール１１は、洗練された抽象構文木を受け取り、これをＨＤＬによる回路の記述に変換する。この段階では、抽象構文に残された各構成が、良好に特定されたハードウェアインプリメンテーションを有する。その一部を以下に説明する。全般的な技術は、以下に示す重要な点において、公知の技術とは異なっている。

【０１１９】（１）　アサインメントが、より複雑なものであり得るとともに、チャネル通信及びパラメータ化されたファンクションコールを含み得る。そのためには、例えば乗算器にその引数がいつ準備状態にあるかが知らされるように、表現の部分間のより複雑なプロトコルが要求される。

【０１２０】（２）　コンパイラが、ネットリストではなくＨＤＬを生成するので、幾分かの選択の余地がある。具体的には、設計チェーンにおけるより低い統合ツールまで、状態マシン及びレジスタを実行する方法に、選択の余地がある。

【０１２１】基本的なスキームは、抽象構文内の制御ステートメントからの状態マシンとして、制御パスを統合することである。殆どの状態において、何らかの計算が行われる。状態マシンは、計算を初期化し、その計算が完了するのを待ってから次のステージに進む。例えばＩＦの場合、次の状態の位置は、計算された値に依存する。１つの状態マシンが、その後に同時に実行される他の１セットの状態マシンを起動し得るようにすることによって、並列処理が行われる。

【０１２２】それぞれの計算が完了するまでにかかり得る時間は未知であるので、例えば、チャネル或いは外部装置との送信或いは受信が行われる場合、データ依存型の計算が行われる場合、或いは、ファンクションコールが行われる場合には、その計算を実行する回路は、計算の完了を信号で知らせるとともに、その値が使用されるの待ってからその値をディスアサートできなければならない。これは、以下に述べる表現のプロトコルの複雑さを説明する。最適化工程によって抽象構文が十分に単純化されるならば、より単純なプロトコルを用いることも可能であるが、その場合、実行時間が長くなり得る。さらに、統合後のゲートレベルの何らかの最適化は、必要とされないシグナリングの余分なレベルを除去する。

【０１２３】図２Ａ～図２Ｃは、制御パス用の基本的なビルディングブロックがどのように形成されるのかを示す図である。ステージは、抽象構文木によって決まる形状に従う。図２Ａには、１プロセスを表す状態マシン１５が示されている。大きい円１６は制御ノードであり、各制御ノードは、１つ或いは１セットのアクションに関連し得る。最も単純なケースにおいて、これらの制御ノードは、抽象構文言語内の基本プロセスに対応するアサインメント或いは通信である。状態マシン１５は、そのアクションが完了してからでないと、次の状態に進むことができない。

【０１２４】図２Ｂには、コンポーネント状態マシン１７及び１８のシーケンシャル構造として、シーケンシャル構造が示されており、第１のマシンの終了状態１９が第２のマシンの開始状態と１つになっている。

【０１２５】図２Ｃには、並列構造が示されている。一つのマスタープロセス２０は特別なものであり、通常の方法によって現在のシーケンシャル状態マシンに挿入される。他の全てのスレーブプロセス２１は、マスタープロセスが開始するのを待ってから開始する。

【０１２６】並列部分の終端では、マスタープロセス２０は、全てのスレーブプロセス２１が終了するのを待ってから、次に進む。その後、各スレーブプロセス２１は初期待機状態に戻って、次の起動に備える。実行中は、マスタープロセス及びスレーブプロセスのステータスは同じである。つまり、両者を区別するのは、その開始方法だけである。

【０１２７】ａ及びｂが幅８で宣言されたものとして、

【０１２８】

【数２７】

```
Define IDI("a")
Define 1D2("b")
a := 1
b := a + b
```

【０１２９】という抽象構文のフラグメントを考える。

【０１３０】図３は、このプログラムのために生成され得る可能な回路例を示す。図３の左側には、この例における抽象状態マシンが示されている。各アサインメントにつき１つ、合計２つの中間状態２２があり、また、通常の開始状態２３及び終了状態２４がある。図３の残りの部分は、可能な回路を示す。フリップフロップ２５、２６、２９及び３０は、グローバルクロック（不図示）に接続され、立ち上がりエッジで起動する。

【０１３１】リセット付きＤ型フリップフロップ２５及び２６は、「ワンホット」エンコードにおける状態マシンを表す。これは、各フリップフロップが可能な状態の１つを表していることを意味する。フリップフロップが１を有するときには状態はアクティブであり、そうでないときには非アクティブである。プログラムの開始前にフリップフロップ２５及び２６を０に設定するために、リセットライン２７が必要である。この他のエンコードを用いて状態マシンを表すことも可能であるが、この例が恐らく最も単純である。

【０１３２】開始パルス２８は、１クロックサイクル毎に１状態の割合で、チェーン上を移動する。これは、特別な例である。なぜなら、各アサインメントは１サイクルしか要しないとが仮定されているからである。より複雑な例の場合、関連アクションが完了するまでパルスを待機させるために、何らかの回路機構を生成する必要がある。

【０１３３】第１の中間状態においては、変数ａのためのレジスタ２９のイネーブルビットは真に設定されており、これにより、次の立ち上がりクロックエッジにおいて、８ビットの定数値１（２進数で００００００００１）を格納することが可能になる。

【０１３４】第２の中間状態においては、変数ｂのレジスタ３０のイネーブルビットは、シングルサイクル加算器３１のイネーブルビットと同様に真に設定されている。従って、ａ及びｂの以前の値は、次の立ち上がりクロックエッジが生じたときにｂに格納される。

【０１３５】図４Ａは、Ｒ表現３２（即ち、その値がデータとして要求されている表現）がどのようにインターフェースされるかを示す。Ｒ表現の値が要求されると、信号Ｒｒｅｑｕｅｓｔがアサートされる。上記の値が要求されなくなるまでの間、信号Ｒｒｅｑｕｅｓｔを真に保っておかなければならない。その後、この表現はある値を計算し、その値を信号Ｒｖａｌｕｅとして出力するとともに、その信号Ｒｖａｌｕｅが有効であることを示すために信号Ｒｒｅａｄｙをアサートする。Ｒｖａｌｕｅが要求されなくなると、入力信号ｇｏが、１クロックサイクルだけ真となり、そして信号Ｒｒｅｑｕｅｓｔは偽になる。信号Ｒｒｅｑｕｅｓｔが次に真となるまで、信号Ｒｖａｌｕｅ及びＲｒｅａｄｙはディスアサートされる。無論、定数及び組合せ表現等の多くの単純な表現の場合、本スキームにおける明らかな複雑さの大部分は、統合中にゲートレベル最適化器によって容易に単純化され得る。

【０１３６】図４Ｂにおいては、Ａ＋ＢについてのＲ表現が、Ａ及びＢについてのＲ表現３３及び３４、加算器３５、及び組合せロジックから構成される。Ｒ表現は、定数、単純変数、アレイのリファレンス、チャネル入力、或いは外部メモリからの読出し、ならびに通常の演算的、論理的及びビット的組合せであり得る。Ｒｒｅｑｕｅｓｔ及びｇｏ信号は、両方のコンポーネント３３及び３４に一斉送信され、そのＲｖａｌｕｅは加算器３５に与えられる。この例において、加算器３５は組合せ加算器であると仮定されている。複合物のＲｒｅａｄｙとして、ＲｒｅａｄｙのブールＡＮＤをとる。この回路は、加算器３５を必要な関数に変えるだけで、あらゆる

組合せ表現を実行するのに十分なものとなる。実行する処理が組合せではない場合、演算子自体が、適切な方法によって２つのコンポーネントのＲｒｅａｄｙを組み合わせることによって、Ｒｒｅａｄｙを提供しなければならない。このような接続は、パワー削減のためにも利用され得る。その場合、入力データが有効になるまで、加算器３５をオフにする。

【０１３７】オペランドと演算子入力との間に幾つかのマルチプレクサを挿入し、演算子出力の上にデマルチプレクサを挿入するだけで、１つの加算器（或いは他の演算子）を、いくつかの計算によって共有することが可能になる。最適化器モジュール８がコンフリクトが全く無いこと、例えば一度に２つの計算が同一の演算子を使用しようとしていないことを、確かめる必要がある（これは、スケジューリング及びアロケーションと呼ばれる）。

【０１３８】図５Ａ～図５Ｃに、その他のＲ表現を示す。図５Ａは、コール・バイ・バリュー関数がどのようにコールされ得るのかを示す。実際のパラメータ（引数）を連結することによって、１つの表現Ｒ３８が与えられる。この表現は、準備状態になると、プロセスを始動する。このプロセスは、図４Ａ及び図４Ｂのスレーブプロセスと同様に関数Ｆ３９を実行する。Ｆが一度に２回以上起動されないようにするためにＦにとって必要な調停は、図５Ａには図示されておらず、この単純なロジックは、Ｆの内部に設けられている。Ｆからの全ての戻り値は、Ｒｖａｌｕｅ信号を介して発信側に渡され、また、Ｆが終了すると、Ｒｒｅａｄｙがアサートされる。ｇｏ信号は、Ｆ及びＲに一斉送信される。

【０１３９】図５Ｂは、単純変数をどのように実施するのかを示す。値自体はレジスタ（不図示）に格納されており、そのレジスタからの出力は、Ｒｖａｌｕｅ信号を介して、その出力を要求する各Ｒ表現に対して利用可能になっている。Ｒｒｅｑｕｅｓｔ及びｇｏ信号は無視される。この値は常に利用可能であるので、Ｒｒｅａｄｙはロジック１に固定されている。

【０１４０】図５Ｃは、チャネルがどのように読み出されるのかを示す。チャネルのｔｘｒｅａｄｙ信号が真であれば、その表現は準備状態である。ハンドシェークの最終部は、ｇｏ信号である。特定のチャネルから読出しを行う全てのＲ表現からのｇｏ信号のＯＲをとることにより、そのチャネルのｒｘｒｅａｄｙ（受信準備完了）信号を生成する。

【０１４１】図６Ａ及び図６Ｂは、どのようにしてＬ表現（値のデスティネーションを表す表現）を作成するのかを示す。図６Ａは、Ｌ表現４２のための標準的なインターフェースを示す。Ｌ表現は、単純変数、アレイのリファレンス、チャネルの出力、外部メモリへの書込み、或いはそれらの組合せであり得る。Ｒｒｅｑｕｅｓｔ信号を用いて、Ｌ表現内のあらゆる埋込みＲ表現（通常は、アレイのインデックス計算）を開始する。Ｌｒｅｑｕｅｓｔ信号は、真性Ｌ表現を開始し、また、Ｌｖａｌｕｅ信号に有効なデータが存在するときには真に設定される。格納処理の完了準備が整うと、ＬＲｒｅａｄｙ信号が立ち上がる。最後に、表現の環境が準備状態になったとき、１サイクルの間ｇｏ信号を真にすることによって、リソースの解放を示す。Ｌ表現の組合せの場合、２クロックサイクルを要求できるのはＬ表現の中の１つだけであり、そして、このサブ表現が、処理全体のタイミングを決定する。他の全てのサブ表現のＬＲｒｅａｄｙは、常に真でなければならない。この条件が満たされない場合、プロトコルが失敗し得る。

【０１４２】図６Ｂにおいては、アサインメントがどのように構築されるのかを説明するために、インターフェースが用いられている。

【０１４３】図７Ａ～図７Ｃは、いくつかの特定のＬ表現がどのようにしてエンコードされるのかを示す。図７Ａは、単純変数（レジスタ）に対する書込みがどのように行われるのかを示す。書込みデータを、３状態ドライバ４４を介して、その変数用の書込みバスの上に流す。３状態ドライバ４４は、信号ｇｏが送信されるとイネーブルされる。これがうまく作動するように、書込みに要するサイクル数を１とする。そのレジスタの書込みイネーブル信号をとって、そのレジスタに書込みを行う全てのＬ表現に対する全ての書込みイネーブル信号の論理ＯＲにする。コンフリクトが全く生じ得ないようにするのは、最適化ステージまでである。

【０１４４】図７Ｂは、外部メモリへの書込みがどのように行われるのかを示す。ｇｏ信号が到達するまでの間、書込み完了信号を真に保っておかなければならない。やはり、このメモリ装置に関係する全ての書込みイネーブルのＯＲがとられなければならない。

【０１４５】図７Ｃは、チャネル出力がどのように行われるのかを示す。所与のチャネルに対するチャネル出力の全てのＬ表現が集められる。そのチャネルのｔｘｒｅａｄｙ（発信準備完了）は、（このチャネルに言及する各Ｌ表現に１つずつある）部分的ｔｘｒｅａｄｙ信号の全てのＯＲである。個々のｒｘｒｅａｄｙ信号は、チャネルｒｘｒｅａｄｙに直接に接続される。

【０１４６】図８は、ｉｆ ｂ ｔｈｅｎ Ｐ ｅｌｓｅ Ｑのインプリメンテーションを示す。Ｒ表現ｂからのｒｅａｄｙ信号は、マルチプレクサによって方向づけられる。このマルチプレクサは、ｂが返した値によって制御される。これにより、状態マシンがＰまたはＱのどちらを伴って継続するかが選択される。

【０１４７】図９は、ｗｈｉｌｅ Ｃ（ｂ） ｄｏ Ｐのインプリメンテーションを示す。状態マシンは、ｂの値によって、再びＰを実行するか、或いは、Ｐを実行せずに次に進むように指示される。

【０１４８】図１０Ａ及び図１０Ｂは、リソースの作成

方法を示す。変数、アレイ、チャネル、或いは関数のそれぞれが、リソースである。ＨＤＬ生成器モジュール１１が構文木を詳細に検討した後の時点では、各リソースは、１つ以上の様々なＲ表現及びＬ表現によってアクセスされている。各リソースについて、ＨＤＬ生成器モジュール１１は、これらのＲ表現及びＬ表現の「バックエンド」からの信号を用いて、リソースの正しい挙動を規定する適切な回路を作成しなければならない。

【０１４９】図１０Ａは、書込みイネーブルを有するエッジトリガレジスタ４５として実施される、単純変数をどのように形成するのかを示す。Ｌ表現からの（書き込まれるべき値を持つ）データバスは結合され、書込みイネーブル信号は一緒にＯＲをとられる。出力（Ｒ表現）は比較的簡単であり、必要とされるところにデータがコピーされるだけである。

【０１５０】図１０Ｂは、どのようにチャネルが形成されるのかを示す。全てのＬ表現（チャネル出力）は、それぞれデータバスを有する。それらのデータバスは結合され、このチャネルの全てのＲ表現（チャネル入力）のデータバスにコピーされる。チャネルの読出し箇所はｍ箇所あり、また、チャネルの書込み箇所がｎ箇所あると仮定されている。図１０Ｂの４６において、書込み側のｔｘｒｅａｄｙ信号のＯＲをとり、これを読出し側に一斉送信する。同様に、図１０Ｂの４７において、読出し側のｒｘｒｅａｄｙ信号のＯＲをとり、これを書込み側に一斉送信する。

【０１５１】（例）

【０１５２】

【数２８】

```
void main()

{

    unsigned #8 x;
    for (x=0; x<10; x++)
        {
        x<<=1;
        }

}
```

【０１５３】というフラグメント例を考える。

【０１５４】これは、非常に不自然な例である。なぜなら、入力も出力もないからである。しかし、この例は、短くて理解し易い。変数Ｘは、値０から始まる。その後、変数Ｘをインクリメントして、これを左に１ビットシフトし、その工程を変数Ｘが１０未満でなくなるまで反復する。その後に、プログラムを終了する。この時点では、出力が多少整理されており、可読性が高められている。

【０１５５】記号テーブルは、

【０１５６】

【数２９】

```
ID1 FUNCTION main void  → void
        1D2 VARIABLE x      int#8
```

【０１５７】というエントリを有する。

【０１５８】抽象構文は、

【０１５９】

【数３０】

```
Define ID1("main")
  Declare ID2("x")
    FOR (x=0(#8); x<10(#8); x++)
    x<<=1;
```

【０１６０】となる。

【０１６１】ＨＤＬ生成器には「ｆｏｒ」構成がなく、副次的効果を持つアサインメントは、その副次的効果を明示しなければならない。従って、上記の例は、

【０１６２】

【数３１】

```
Define ID1("main")
  Declare ID2 ("x")
  x   := 0(#8)
  While (x < 10(#8))
      x   := x << 1
      x   := x + 1(#8)
```

【０１６３】のように単純化される。

【０１６４】最適化器モジュールは、ループ内の２つのアサインメントが組み合わされ得ることを発見する。

【０１６５】

【数３２】

```
Define ID1 ("main")
  Declare ID2 ("x")
  x     := 0(#8)
  While (x < 10(#8))
    x  := (x << 1) + 1(#8)
```

【０１６６】最終的に、ＨＤＬ生成器モジュールは、統合用のＶＨＤＬ　ＲＴＬで書かれた以下の出力を生成する。初めにエントリ宣言があり、これにより、周辺とのインターフェースが記述される。

【０１６７】

【数３３】

25

```
library ieee;
use ieee. std_logic_1 164. all;
use ieee . std_logic_arith. all;
use work.bach_std.all;
entity main is
        port(
            cik : in std_logic;
            reset : in std_logic;
            start : in std_logic;
            finish : out std_logic
           );
end main;
```

26

【０１６８】第２に、アーキテクチャがあり、これにより、エントリの挙動が記述される。これは、３つの部分、即ち、いくつかのローカル宣言と、制御パスのための状態マシンと、格納場所及びデータパスのためのレジスタ定義とに分かれる。

【０１６９】ローカル宣言

【０１７０】

【数３４】

```
type sm_main_O_type is (state_main_O_O, state_main_O_1,
        state_main_O_3, state_main_O_4);
signal sm_main_O : sm_main_O_type;
signal var_x3 : unsigned (7 downto 0);
signal varw_x3 : unsigned (7 downto 0);
begin
```

【０１７１】メイン(main)のアーキテクチャＲＴＬを、以上に示す。

【０１７２】制御パス状態マシンは、

【０１７３】

【数３５】

```
main_0: process (clk)
begin
    if ((clk'event and clk='1')) then
        if ((reset = '1')) then
            sm_main_0 <= state_main_0_0;
        else
            case (sm_main_0) is
                when state_main_0_0 =>
                    if ((start = '1')) then
                        sm_main_0 <= state_main_0_1;
                    end if;
                when state_main_0_1 =>
                    sm_main_0<=state_main_0_3;
                when state_main_0_3 =>
                        if ((var_x3>=unsigned ' ("00001010"))) then
                            sm_main_0<=state_main_0_0;
                        elsif ((var_x3<unsigned ' ("00001010"))) then
                            sm_main_0<=state_main_0_4;
                        end if;
                when state_main_0_4 =>
                        sm_main_0<=state_main_0_3;
            end case;
        end if;
    end if;
        end process;
```

【０１７４】のように示される。

【０１７５】格納場所及びデータパスは、

【０１７６】

【数３６】

```
    var_x3 <= varw_x3;
    proc_var_x3: process (clk)
    begin
        if ((clk'event and clk = '1')) then
            if ((reset = '1')) then
                varw_x3 <= conv_unsigned(0, 8);
        else
                if ((sm_main_0 = state_main_0_4)) then
                    varw_x3 <= conv_unsigned(((var_x3 * 2)
                                + unsigned ' ("00000001")), 8);
        elsif ((sm_main_0 = state_main_0_1)) then
                varw_x3 <= conv_unsigned(unsigned ' ("00000000"), 8);
        end if;
      end if;
        end if;
    end process;
    finish <= boo12bit((sm_main_0 = state_main_0_0));
end RTL;
```

【０１７７】のように示される。

【０１７８】

【発明の効果】以上に説明したように、本発明によれば、集積回路の設計にあたって、Ｃ言語に類似した言語によって記述された集積回路の機能や仕様などに関するソフトウェアアルゴリズムが、高レベルの最適化を行う適切なコンパイラの使用によって、自動的に或いは半自動的に、ハードウェア記述言語（ＨＤＬ）に変換（コンパイル）される。ソフトウェアアルゴリズムを記述する言語としては、並列処理や同期通信を記述できる高レベル言語（例えば、並列Ｃ言語）を使用することができる。従って、本発明によれば、集積回路の設計にあたって、高レベル言語で記述されたソフトウェアレベルからハードウェアレベルへの変換時間が短縮され、ハードウェア開発の効率が向上する。

【図面の簡単な説明】

【図１】本発明のある実施形態の一部を構成するハードウェアコンパイラの構造を示す概略図である。

【図２Ａ】図１のコンパイラによって制御パスがどのように統合されるのかを概略的に示す図であり、特に、開始及び終了時間が１つである単一の処理を説明するための図である。

【図２Ｂ】図１のコンパイラによって制御パスがどのように統合されるのかを概略的に示す図であり、特に、２つの処理をシーケンシャルに実行する方法を説明するための図である。

【図２Ｃ】図１のコンパイラによって制御パスがどのように統合されるのかを概略的に示す図であり、特に、数個の処理を同時に実行する方法を説明するための図である。

【図３】簡単なプログラム例を実行するための可能な回路例を示す図である。

【図４Ａ】表現をどのようにエンコードするのかを説明するための図であり、特に、単一のＲ表現の場合を説明するための図である。

【図４Ｂ】表現をどのようにエンコードするのかを説明するための図であり、特に、Ａ及びＢの表現から表現Ａ＋Ｂをどのように作成するのかを説明するための図である。

【図５Ａ】コール・バイ・バリューファンクションコールがどのように行われるのかを説明するための図である。

【図５Ｂ】単純変数がどのように読み出されるのかを説明するための図である。

【図５Ｃ】チャネルがどのように読み出されるのかを説明するための図である。

【図６Ａ】Ｌ表現がどのようにエンコードされるのかを示す図であり、特に、単一のＬ表現インターフェースを説明するための図である。

【図６Ｂ】Ｌ表現がどのようにエンコードされるのかを示す図であり、特に、どのようにＬ表現とＲ表現を組み合わせてアサインメントを生成するのかを説明するための図である。

【図７Ａ】特定のＬ表現がどのようにエンコードされるのかを示す図であり、特に、単純変数或いはレジスタへの書込みがどのように行われるのかを説明するための図である。

【図７Ｂ】特定のＬ表現がどのようにエンコードされるのかを示す図であり、特に、外部メモリへの書込みがどのように行われるのかを説明するための図である。

【図７Ｃ】特定のＬ表現がどのようにエンコードされるのかを示す図であり、特に、チャネル出力がどのように行われるのかを説明するための図である。

【図８】条件ステートメントをどのように作成するのかを説明するための図である。

【図９】ループステートメントをどのように作成するのかを説明するための図である。

【図１０Ａ】どのようにリソースを作成するのかを説明するための図であり、特に、単純変数をどのように作成するのかを説明するための図である。

【図１０Ｂ】どのようにリソースを作成するのかを説明するための図であり、特に、チャネルをどのように作成

29

するのかを説明するための図である。

【符号の説明】

１　ソースコード

５　記号テーブル

６　単純化器モジュール

30

８　最適化器モジュール

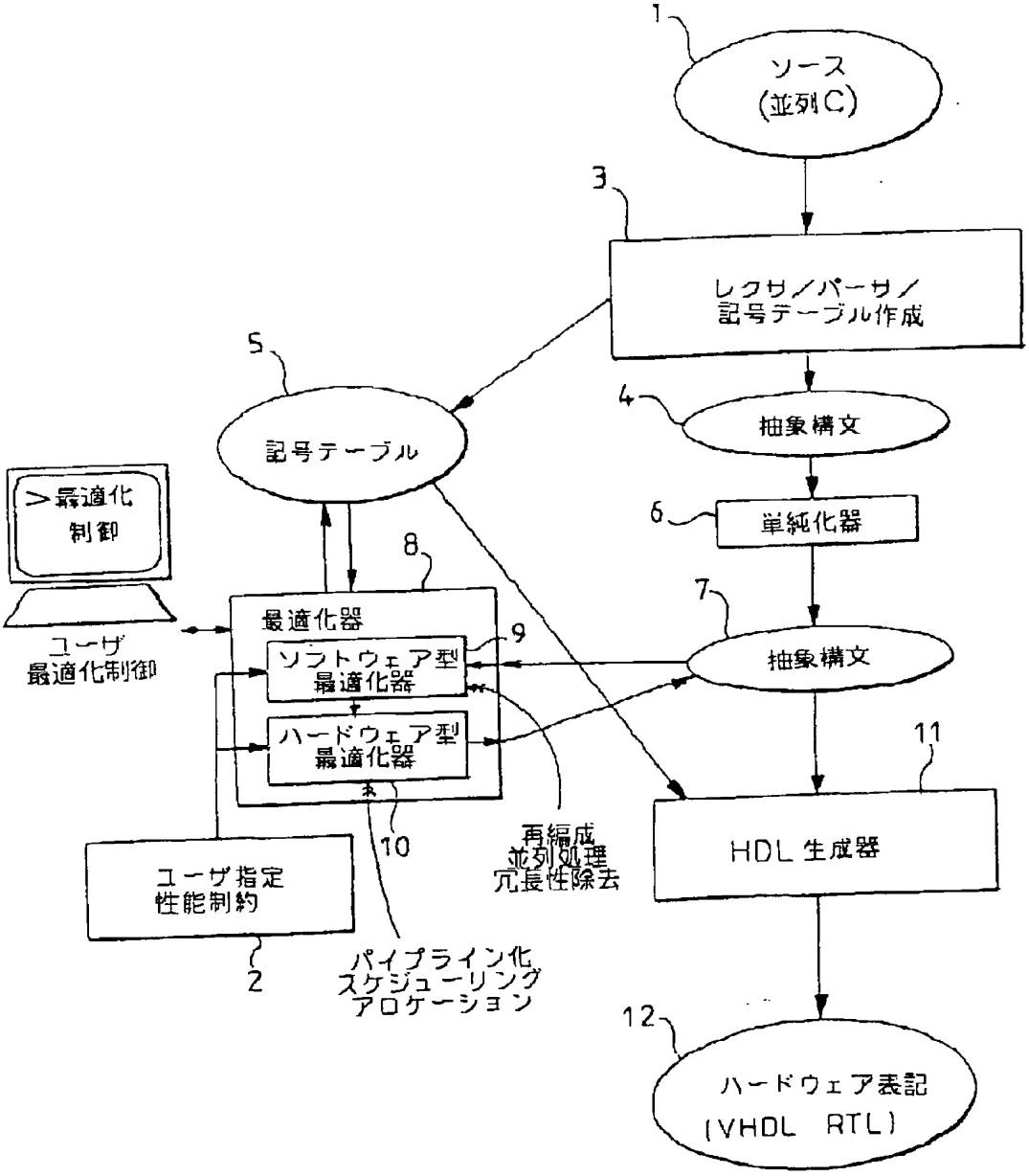
９　ソフトウェア最適化器モジュール

１０　ハードウェア最適化器モジュール

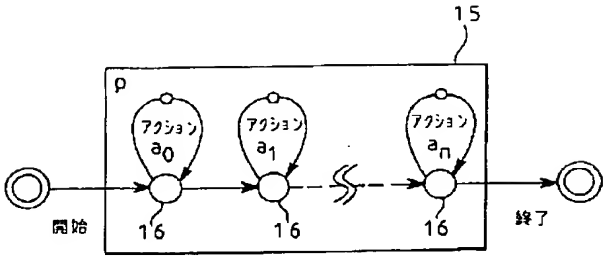
１１　ＨＤＬ生成器モジュール

１２　ハードウェア表記（出力コード）

【図１】

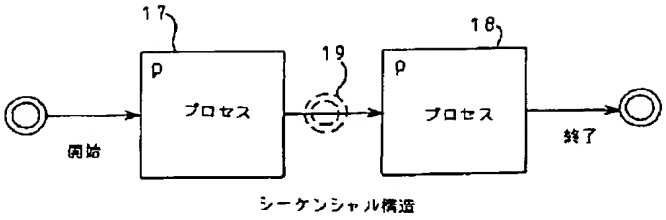


【図２Ａ】



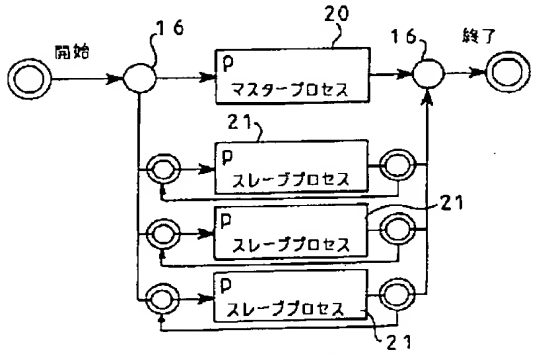
プロセス始動

【図２Ｂ】



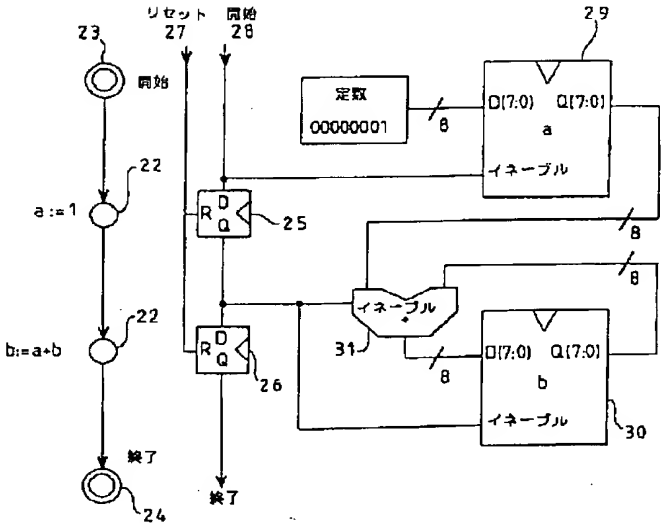
シーケンシャル構造

【図２Ｃ】

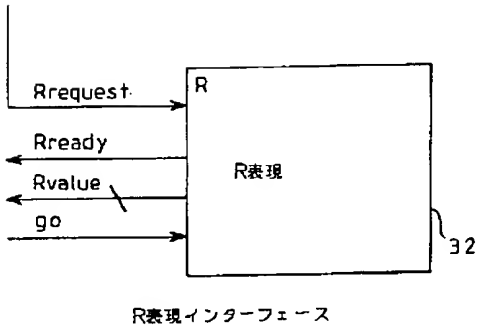


パラレル構造

【図３】

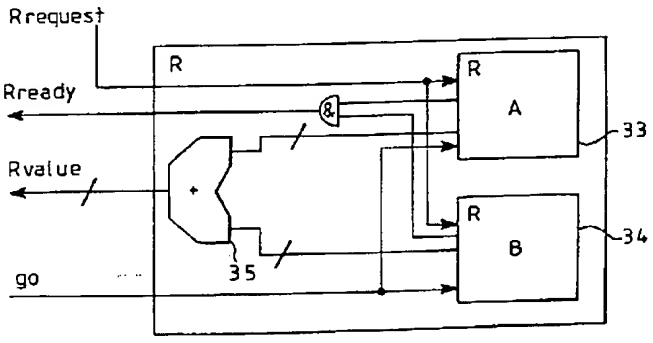


【図４Ａ】



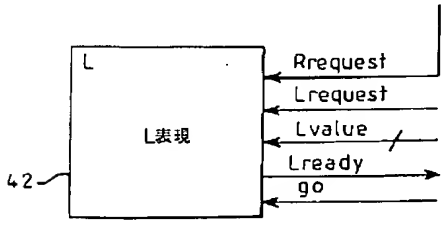
R表現インターフェース

【図４Ｂ】



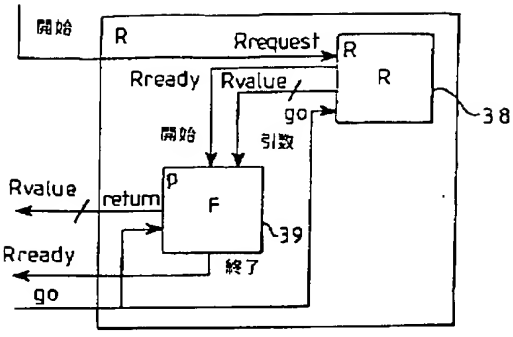
「A+B」の構造

【図６Ａ】



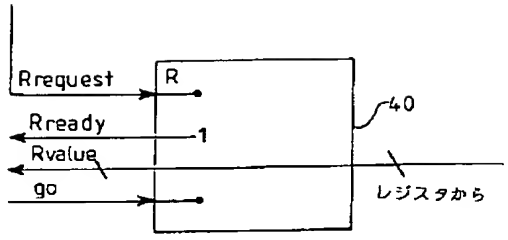
L表現インターフェース

【図５Ａ】



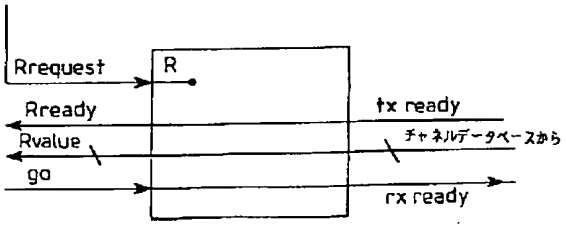
コール・バイ・バリュー関数 f(R)

【図５Ｂ】



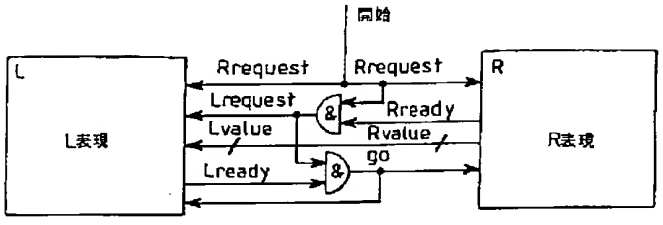
単純変数へのR表現インターフェース

【図５Ｃ】



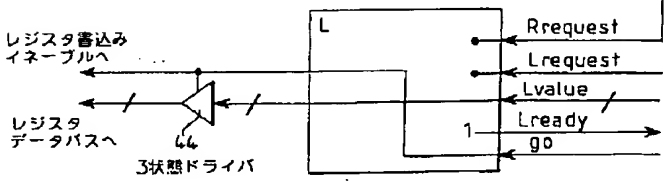
チャネルへのR表現インターフェース

【図６Ｂ】



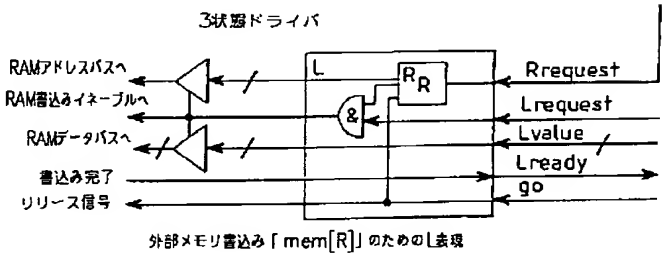
アサインメント(L:=R)

【図７Ａ】



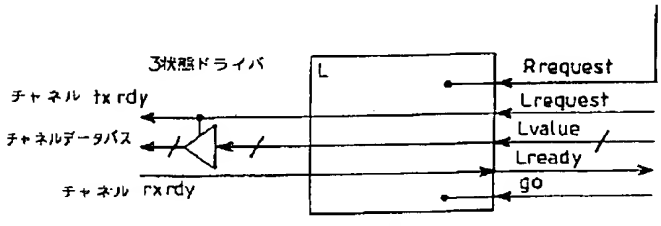
単純変数のためのL表現

【図７Ｂ】



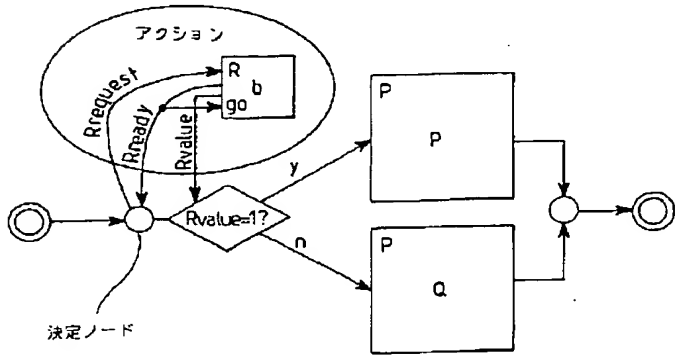
外部メモリ書込み「mem[R]」のためのL表現

【図７Ｃ】

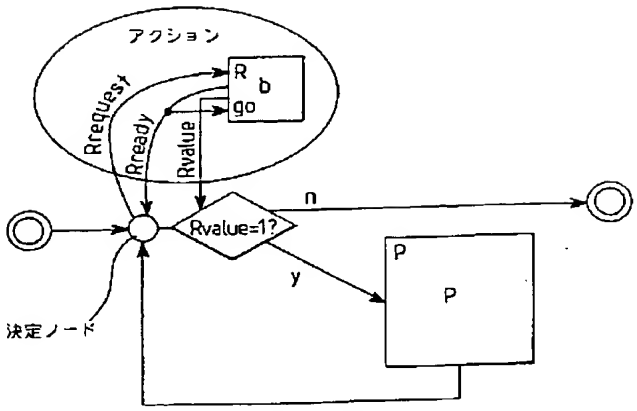


チャネル出力のためのL表現

【図８】

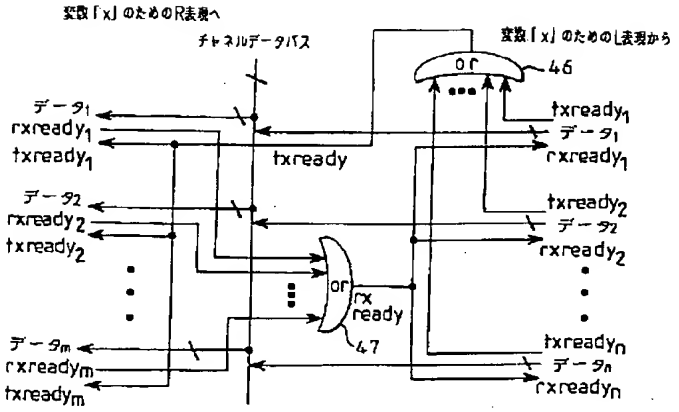


【図９】



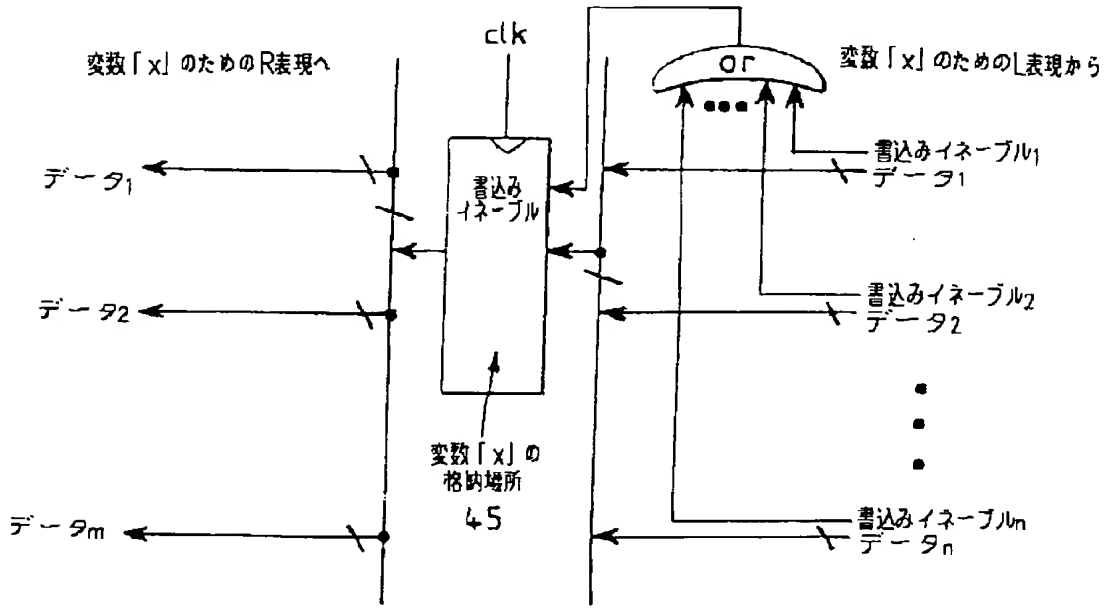
表現と制御との組合せ

【図１０Ｂ】



チャネル形成

【図１０Ａ】



単純変数の形成